Data Structures Exam (Intermediate)

Instructions:

- This exam consists of 50 questions.
- Answer all questions to the best of your ability.
- Choose the best answer for Multiple Choice Questions (MCQ).
- Indicate whether the statement is True or False.
- For tracing questions, show your work and provide the final output.

Section 1: Multiple Choice Questions (MCQ)

1. What is the primary purpose of a data structure? a) To encrypt data for security. b) To organize data items by considering their relationship to each other. c) To convert data from one type to another. d) To compress data to save storage space.

2. Which of the following is a measure of program efficiency related to the amount of memory a program requires? a) Time complexity b) Space complexity c) Algorithmic complexity d) Execution time

3. An Abstract Data Type (ADT) defines the: a) Specific implementation details of a data structure. b) Interface of an organized data object and its operations. c) Memory allocation strategy for data. d) Graphical representation of data.

4. Which of the following is a characteristic of an array? a) It is a non-primitive, non-linear data structure. b) It stores items of different types contiguously in memory. c) It stores items of the same type contiguously in memory. d) Its size can be dynamically changed without re-allocation.

5. In C, if you declare `int A[10];`, what does `A[3]` refer to? a) The third element of the array `A`. b) The fourth element of the array `A`. c) The size of the array `A`. d) The starting address of the array `A`.

6. Which of the following operations is typically associated with a stack? a) Enqueue b) Dequeue c) Push d) Remove

7. A stack is often described as a data structure that follows the principle of: a) FIFO (First-In, First-Out) b) LIFO (Last-In, First-Out) c) FILO (First-In, Last-Out) d) LILO (Last-In, Last-Out)

8. What is the primary application of a stack in managing function calls? a) To store return values of functions. b) To manage the order of function execution and local variables. c) To optimize function call overhead. d) To store global variables.
9. If a stack is implemented using an array, what is the condition for an overflow? a) When the stack is empty and a `pop` operation is attempted. b) When the array is full and a `push` operation is attempted. c) When the array is empty and a `push` operation is attempted. d) When the stack contains only one element.
10. Which operation is used to add an element to the rear of a queue? a) Pop b) Push c) Enqueue d) Dequeue
11. A queue is a data structure that operates on the principle of: a) LIFO (Last-In, First-Out) b) FIFO (First-In, First-Out) c) FILO (First-In, Last-Out) d) LILO (Last-In, Last-Out)
12. Which of the following is a common application of queues? a) Managing recursive function calls. b) Implementing undo/redo features. c) CPU scheduling in operating systems. d) Evaluating arithmetic expressions.
13. What is the time complexity for adding an element to a queue implemented using a linked list, assuming both front and rear pointers are maintained? a) O(n) b) O(log n) c) O(1) d) O(n^2)
14. In a singly linked list, each node typically contains: a) Data and a pointer to the previous node. b) Data and a pointer to the next node. c) Data, a pointer to the previous node, and a pointer to the next node. d) Only data.
15. Which of the following is an advantage of linked lists over arrays for dynamic data storage? a) Faster random access to elements. b) Less memory overhead per element. c) Dynamic size and efficient insertions/deletions anywhere. d) Better cache locality.
16. To insert a new node at the end of a singly linked list, what pointer needs to be updated, assuming you have a pointer to the tail? a) The head pointer. b) The `next` pointer of the current tail node. c) The `next` pointer of the new node. d) No pointers need to be updated.
17. What is the time complexity of deleting the head node from a singly linked list? a) O(n) b) O(log n) c) O(1) d) O(n^2)
18. In a linked stack, where are elements typically added and removed from? a) The middle of the list. b) The tail of the list. c) The head of the list. d) Both head and tail.

19. What is the main advantage of a linked stack over an array-based stack? a) Better memory utilization (no wasted space). b) Faster random access. c) Fixed size. d) Simpler implementation.

20. In a tree data structure, a node that has no children is called a: a) Root node. b) Internal node. c) Leaf node. d) Parent node.

21. The root node of a tree is the node that: a) Has no children. b) Has exactly one child. c) Has no parent. d) Is at the lowest level of the tree.

22. What is the maximum number of nodes at level `L` in a binary tree (assuming root is at level 0)? a) L b) 2L c) 2^L d) L+1

23. Which tree traversal method visits the root node first, then the left subtree, then the right subtree? a) In-order traversal. b) Pre-order traversal. c) Post-order traversal. d) Level-order traversal.

24. In a Binary Search Tree (BST), for any given node, all values in its right subtree are: a) Less than the node\'s value. b) Equal to the node\'s value. c) Greater than the node\'s value. d) Unordered.

25. Which traversal of a BST produces elements in ascending sorted order? a) Pre-order traversal. b) Post-order traversal. c) In-order traversal. d) Level-order traversal.

26. What is the worst-case time complexity for searching an element in a skewed Binary Search Tree? a) O(1) b) O(log n) c) O(n) d) O(n log n)

27. An AVL tree is a self-balancing binary search tree where the height difference between the left and right subtrees of any node is at most: a) 0 b) 1 c) 2 d) log n

28. What is the primary purpose of rotations in an AVL tree? a) To increase the height of the tree. b) To maintain the balance property of the tree. c) To sort the elements in the tree. d) To reduce the number of nodes.

29. What is a collision in the context of hashing? a) When a key cannot be inserted into the hash table. b) When two different keys produce the same hash value. c) When the hash table is completely full. d) When a key is not found during a search operation.

30. Which of the following is a common collision resolution technique? a) Binary search. b) Linear probing. c) Merge sort. d) Quick sort.

31. The load factor of a hash table is calculated as: a) (Table size) / (Number of elements) b) (Number of elements) / (Table size) c) (Number of collisions) / (Table size) d) (Number of empty slots) / (Number of elements)
32. What is the primary goal of a good hash function? a) To be computationally expensive. b) To produce many collisions. c) To distribute keys uniformly across the hash table. d) To ensure all keys map to the same index.
33. Which of the following data structures is best suited for implementing a dictionary or map? a) Stack b) Queue c) Linked List d) Hash Table
34. What is the time complexity of searching for an element in a hash table using chaining, on average? a) O(n) b) O(log n) c) O(1) d) O(n log n)
35. In a hash table using open addressing, what happens when a collision occurs? a) A new linked list is created at that index. b) The colliding element is discarded. c) The algorithm probes for the next available slot. d) The hash table is resized immediately.

Section 2: True/False Questions

1. Space complexity refers to the amount of time a program takes to run. (True/False)
2. An ADT (Abstract Data Type) specifies how data is stored and manipulated internally. (True/False)
3. In a stack, the `peek` operation removes the top element. (True/False)
4. A queue is a LIFO (Last-In, First-Out) data structure. (True/False)
5. A singly linked list allows efficient traversal in both forward and backward directions. (True/False)
6. Deleting a node from the middle of a singly linked list requires traversing the list from the beginning. (True/False)
7. A linked queue can suffer from overflow if memory is exhausted. (True/False)
8. The `top` pointer in a linked stack always points to the last element inserted. (True/False)
9. A binary tree can have at most two children per node. (True/False)

10. In a complete binary tree, all levels are completely filled except possibly the last level, which is filled from left to right. (True/False)
11. Pre-order traversal of a BST always produces elements in ascending order. (True/False)
12. Deleting a node from a BST always requires rebalancing the tree. (True/False)
13. AVL trees guarantee O(log n) time complexity for search, insertion, and deletion operations. (True/False)
14. A hash function should distribute keys evenly across the hash table. (True/False)
15. Chaining is a collision resolution technique that uses linked lists to store elements that hash to the same index. (True/False)

Section 3: Tracing Questions

Tracing Question 1: Stack Operations

Consider an empty stack. Perform the following operations and show the state of the stack after each operation. What is the final state of the stack and what is the output of the `pop` operations?

Operations: 1. Push(A) 2. Push(B) 3. Pop() 4. Push(C) 5. Push(D) 6. Pop() 7. Push(E) 8. Pop()

Tracing Question 2: Binary Search Tree Insertion and In-order Traversal

Given an empty Binary Search Tree (BST), insert the following elements in the given order. Then, perform an in-order traversal of the final BST and list the elements in the order they are visited.

Elements to insert: 25, 15, 35, 10, 20, 30, 40, 5

Show the BST after all insertions are complete. Then, list the in-order traversal output.

Data Structures Exam (Intermediate) - Model Answers

Section 1: Multiple Choice Questions (MCQ) - Answers and Explanations

1. **b) To organize data items by considering their relationship to each other.**
    - **Explanation:** As stated in `01Introduction.pptx`, a data structure is "a way of organizing data items by considering its relationship to each other." It's about structuring data for efficient access and modification.
2. **b) Space complexity**
    - **Explanation:** `01Introduction.pptx` explicitly states that "The efficiency of the program depends on two measurements Space complexity Time complexity". Space complexity refers to the amount of memory a program needs to run.
3. **b) Interface of an organized data object and its operations.**
    - **Explanation:** According to `01Introduction.pptx`, an ADT "is defined in terms of the operations that can be performed on instances of the type rather than by how those operations are actually implemented. In other words, an ADT defines the interface of the objects." It focuses on what an ADT does, not how it does it.
4. **c) It stores items of the same type contiguously in memory.**
    - **Explanation:** `01Introduction.pptx` describes an array as "a collection of items –which are of the same type- stored contiguously in the memory." This contiguous storage and homogeneous type are key characteristics.
5. **b) The fourth element of the array `A`.**
    - **Explanation:** In C (and many other languages), array indexing is 0-based. So, `A[0]` is the first element, `A[1]` is the second, and `A[3]` is the fourth element. `01Introduction.pptx` shows `A[3]` calculation as `A+3*sizeof(int)` which points to the fourth element.
6. **c) Push**
    - **Explanation:** As discussed in `02Stack.pptx`, `Push` is the operation to add an element to the top of a stack. `Enqueue` and `Dequeue` are associated with queues, and `Remove` is a general term.

7. **b) LIFO (Last-In, First-Out)**
    - **Explanation:** `02Stack.pptx` clearly defines a stack as a LIFO data structure, meaning the last element added is the first one to be removed.
8. **b) To manage the order of function execution and local variables.**
    - **Explanation:** `02Stack.pptx` (and general computer science principles) highlights that stacks are used for function call management. When a function is called, its activation record (containing local variables, return address, etc.) is pushed onto the call stack. When it returns, the record is popped.
9. **b) When the array is full and a `push` operation is attempted.**
    - **Explanation:** An overflow occurs when you try to add an element to a data structure that has reached its capacity. For an array-based stack, this happens when the underlying array is full and a `push` operation is attempted, as explained in `02Stack.pptx`.
10. **c) Enqueue**
    - **Explanation:** `03Queue.pptx` defines `Enqueue` as the operation to add an element to the rear (or back) of a queue.
11. **b) FIFO (First-In, First-Out)**
    - **Explanation:** `03Queue.pptx` explicitly states that a queue is a FIFO data structure, meaning the first element added is the first one to be removed.
12. **c) CPU scheduling in operating systems.**
    - **Explanation:** `03Queue.pptx` lists CPU scheduling as a common application of queues. Processes waiting for CPU time are often held in a queue, processed in the order they arrived.
13. **c) O(1)**
    - **Explanation:** As shown in `05LinkedStackandQueue.pptx`, if both front and rear pointers are maintained in a linked queue, adding an element (enqueue) involves creating a new node and updating the rear pointer, which takes constant time, O(1).

14. **b) Data and a pointer to the next node.**
    - **Explanation:** `04LinkedList.pptx` illustrates that a node in a singly linked list contains the data and a pointer (or reference) to the next node in the sequence.
15. **c) Dynamic size and efficient insertions/deletions anywhere.**
    - **Explanation:** `04LinkedList.pptx` emphasizes that linked lists offer dynamic memory allocation, allowing them to grow or shrink as needed. Insertions and deletions can be performed efficiently (O(1) if the position is known, or O(n) to find the position) without shifting elements, unlike arrays.
16. **b) The `next` pointer of the current tail node.**
    - **Explanation:** To insert at the end of a singly linked list when you have a tail pointer, you create the new node, set the `next` pointer of the current tail node to point to this new node, and then update the tail pointer to the new node. This is demonstrated in `04LinkedList.pptx`.
17. **c) O(1)**
    - **Explanation:** Deleting the head node in a singly linked list is an O(1) operation. You simply update the `head` pointer to point to the second node in the list, as shown in `04LinkedList.pptx`.
18. **c) The head of the list.**
    - **Explanation:** `05LinkedStackandQueue.pptx` shows that for a linked stack, both `push` and `pop` operations are typically performed at the head of the linked list, as this allows for O(1) time complexity.
19. **a) Better memory utilization (no wasted space).**
    - **Explanation:** Unlike array-based stacks which might pre-allocate a fixed size (leading to wasted space if not fully utilized, or overflow if exceeded), linked stacks allocate memory for each node as needed, thus utilizing memory more efficiently and avoiding fixed-size limitations, as discussed in `05LinkedStackandQueue.pptx`.
20. **c) Leaf node.**
    - **Explanation:** `07Trees.pptx` defines a leaf node as a node that has no children.

21. **c) Has no parent.**

    - **Explanation:** `07Trees.pptx` illustrates that the root node is the topmost node in a tree and is the only node that does not have a parent.

22. **c) 2^L**

    - **Explanation:** `07Trees.pptx` states that the maximum number of nodes at level `L` in a binary tree (with root at level 0) is 2^L.

23. **b) Pre-order traversal.**

    - **Explanation:** `07Trees.pptx` describes Pre-order traversal as visiting the root node first, then recursively traversing the left subtree, and finally the right subtree.

24. **c) Greater than the node\'s value.**

    - **Explanation:** The fundamental property of a Binary Search Tree, as explained in `08BST.pptx`, is that for any given node, all values in its right subtree are greater than the node\'s value.

25. **c) In-order traversal.**

    - **Explanation:** `08BST.pptx` highlights that an in-order traversal (Left-Root-Right) of a BST always produces the elements in ascending sorted order.

26. **c) O(n)**

    - **Explanation:** In the worst-case scenario, a BST can become skewed (like a linked list), where all insertions happen on one side. In such a case, searching for an element might require traversing all `n` nodes, leading to O(n) time complexity, as discussed in `08BST.pptx`.

27. **b) 1**

    - **Explanation:** `09_AVLTree.pptx` defines an AVL tree as a self-balancing BST where the height difference (balance factor) between the left and right subtrees of any node is at most 1 (i.e., -1, 0, or 1).

28. **b) To maintain the balance property of the tree.**

    - **Explanation:** `09_AVLTree.pptx` explains that rotations (LL, RR, LR, RL) are performed in AVL trees to rebalance the tree and maintain the AVL property after insertions or deletions, ensuring logarithmic time complexity for operations.

29. **b) When two different keys produce the same hash value.**
    - **Explanation:** `10_Dictionarys&Hashing.pdf` defines a collision as occurring when two distinct keys map to the same index (or bucket) in a hash table.
30. **b) Linear probing.**
    - **Explanation:** `10_Dictionarys&Hashing.pdf` lists linear probing as a common collision resolution technique, where if a hash collision occurs, the algorithm linearly searches for the next available slot.
31. **b) (Number of elements) / (Table size)**
    - **Explanation:** `10_Dictionarys&Hashing.pdf` defines the load factor (α) as the ratio of the number of elements (n) to the table size (m), i.e., α = n/m.
32. **c) To distribute keys uniformly across the hash table.**
    - **Explanation:** `10_Dictionarys&Hashing.pdf` emphasizes that a good hash function aims to minimize collisions by distributing keys as evenly as possible across the hash table, leading to better performance.
33. **d) Hash Table**
    - **Explanation:** `10_Dictionarys&Hashing.pdf` introduces hash tables as the underlying data structure for implementing dictionaries (or maps) due to their efficient average-case performance for key-value lookups, insertions, and deletions.
34. **c) O(1)**
    - **Explanation:** On average, searching for an element in a hash table using chaining takes O(1) time, assuming a good hash function and a reasonable load factor. In the best case, it\'s O(1), as discussed in `10_Dictionarys&Hashing.pdf`.
35. **c) The algorithm probes for the next available slot.**
    - **Explanation:** In open addressing (including linear probing, quadratic probing, double hashing), when a collision occurs, the algorithm calculates a

new index (probes) to find an alternative empty slot in the hash table to place the element, as described in `10_Dictionarys&Hashing.pdf`.

Section 2: True/False Questions - Answers and Explanations

1. **False**
    - **Explanation:** Space complexity refers to the amount of memory a program requires, while time complexity refers to the amount of time it takes to run, as defined in `01Introduction.pptx`.
2. **False**
    - **Explanation:** An ADT defines the interface and operations of a data structure, but it hides the implementation details (how data is stored and manipulated internally), as explained in `01Introduction.pptx` under "Information hiding (Encapsulation)".
3. **False**
    - **Explanation:** The `peek` operation on a stack allows you to view the top element without removing it. The `pop` operation removes the top element, as detailed in `02Stack.pptx`.
4. **False**
    - **Explanation:** A queue is a FIFO (First-In, First-Out) data structure, meaning the first element added is the first one to be removed. LIFO (Last-In, First-Out) describes a stack, as per `03Queue.pptx`.
5. **False**
    - **Explanation:** A singly linked list only has pointers to the next node, allowing efficient traversal in the forward direction. To traverse backward, you would need a doubly linked list, as shown in `04LinkedList.pptx`.
6. **True**
    - **Explanation:** To delete a node from the middle of a singly linked list, you need to find the node before the one you want to delete so you can update its `next` pointer. This requires traversing the list from the beginning until the predecessor is found, as illustrated in `04LinkedList.pptx`.

7. **True**
    - **Explanation:** While linked queues are dynamically sized, they still rely on available memory. If the system runs out of memory, attempting to allocate a new node for an enqueue operation will result in an overflow (memory exhaustion), as implied by the memory management discussions in `01Introduction.pptx` and linked list concepts in `04LinkedList.pptx`.
8. **True**
    - **Explanation:** In a linked stack, the `top` pointer (or head of the linked list) always points to the most recently added element, which is the one that will be removed next by a `pop` operation, as depicted in `05LinkedStackandQueue.pptx`.
9. **True**
    - **Explanation:** By definition, a binary tree is a tree data structure in which each node has at most two children, referred to as the left and right children, as explained in `07Trees.pptx`.
10. **True**
    - **Explanation:** This is the precise definition of a complete binary tree, as presented in `07Trees.pptx`. All levels are completely filled, except possibly the last level, which is filled from left to right.
11. **False**
    - **Explanation:** Pre-order traversal visits the root first, then the left subtree, then the right subtree. While in-order traversal of a BST produces sorted output, pre-order traversal does not, as explained in `08BST.pptx`.
12. **False**
    - **Explanation:** Deleting a node from a BST does not always require rebalancing. Rebalancing is specifically required in self-balancing BSTs like AVL trees to maintain their balance property after certain insertions or deletions, as discussed in `09_AVLTree.pptx`. A regular BST does not automatically rebalance.

13. **True**
    - **Explanation:** AVL trees are designed to maintain balance, which guarantees that search, insertion, and deletion operations will have a time complexity of O(log n) in the worst case, as detailed in `09_AVLTree.pptx`.
14. **True**
    - **Explanation:** A good hash function aims to distribute keys uniformly across the hash table to minimize collisions and ensure efficient average-case performance, as highlighted in `10_Dictionarys&Hashing.pdf`.
15. **True**
    - **Explanation:** Chaining is a collision resolution technique where all elements that hash to the same index are stored in a linked list at that index, as described in `10_Dictionarys&Hashing.pdf`.

Section 3: Tracing Questions - Answers and Explanations

Tracing Question 1: Stack Operations

Initial state: Empty Stack `[]`

1. **Push(A):**
    - `A` is added to the top of the stack.
    - State: `[A]`
2. **Push(B):**
    - `B` is added to the top of the stack.
    - State: `[A, B]`
3. **Pop():**
    - The top element `B` is removed.
    - State: `[A]`, Output: `B`
4. **Push(C):**
    - `C` is added to the top of the stack.
    - State: `[A, C]`

5. **Push(D):**
    - `D` is added to the top of the stack.
    - State: `[A, C, D]`
6. **Pop():**
    - The top element `D` is removed.
    - State: `[A, C]`, Output: `D`
7. **Push(E):**
    - `E` is added to the top of the stack.
    - State: `[A, C, E]`
8. **Pop():**
    - The top element `E` is removed.
    - State: `[A, C]`, Output: `E`

Final state of the stack: `[A, C]` **Output of the** `pop` **operations:** `B, D, E`

Tracing Question 2: Binary Search Tree Insertion and In-order Traversal

Elements to insert: 25, 15, 35, 10, 20, 30, 40, 5

Building the BST step-by-step:

1. **Insert 25:** (Root) `25`
2. **Insert 15:** (15 < 25, so left child of 25) `25 / 15`
3. **Insert 35:** (35 > 25, so right child of 25) `25 / \ 15 35`
4. **Insert 10:** (10 < 25, 10 < 15, so left child of 15) `25 / \ 15 35 / 10`
5. **Insert 20:** (20 < 25, 20 > 15, so right child of 15) `25 / \ 15 35 / \ 10 20`
6. **Insert 30:** (30 > 25, 30 < 35, so left child of 35) `25 / \ 15 35 / \ / 10 20 30`
7. **Insert 40:** (40 > 25, 40 > 35, so right child of 35) `25 / \ 15 35 / \ / \ 10 20 30 40`
8. **Insert 5:** (5 < 25, 5 < 15, 5 < 10, so left child of 10) `25 / \ 15 35 / \ / \ 10 20 30 40 / 5`

Final BST after all insertions:

```
         25
       /   \
      15    35
    /   \ /   \
   10   20 30   40
  /
 5
```

In-order traversal: (Left -> Root -> Right)

Starting from the root (25): - Go left to 15. - Go left to 10. - Go left to 5. - Visit 5. - Go back to 10. - Visit 10. - Go right from 10 (null). - Go back to 15. - Visit 15. - Go right to 20. - Go left from 20 (null). - Visit 20. - Go right from 20 (null). - Go back to 25. - Visit 25. - Go right to 35. - Go left to 30. - Go left from 30 (null). - Visit 30. - Go right from 30 (null). - Go back to 35. - Visit 35. - Go right to 40. - Go left from 40 (null). - Visit 40. - Go right from 40 (null).

In-order traversal output: 5, 10, 15, 20, 25, 30, 35, 40